ICS: 33.160.25

CCS: M74

UHD World Association
世界超高清视频产业联盟

世界超高清视频产业联盟标准

T/UWA 020-2023

超高清视频处理算法接口技术规范

Technical specification for interface of ultra-high definition video processing algorithm

2023－08－30 发布

2023－08－30 实施

世界超高清视频产业联盟 发布

目　　次

前　言

本文件按照GB/T 1.1-2020《标准化工作导则 第1部分:标准化文件的结构和起草规则》的规定起草。

本文件由世界超高清视频产业联盟（UWA）提出并归口。

本文件起草单位：京东方科技集团股份有限公司、中国电子技术标准化研究院、北京百度网讯科技有限公司、北京广播电视台、北京中联合超高清协同技术中心有限公司、海信视像科技股份有限公司、山东浪潮超高清智能科技有限公司、四川新视创伟超高清科技有限公司、北京奕斯伟科技集团有限公司、华数数字电视传媒集团有限公司、北京流金岁月传媒科技股份有限公司、深圳市奥拓电子股份有限公司、北京市博汇科技股份有限公司、北京锐马视讯科技有限公司、咪咕文化科技有限公司、北京数码视讯软件技术发展有限公司、深圳创维—RGB电子有限公司、杭州当虹科技股份有限公司、上海数字电视国家工程研究中心、深圳市洲明科技股份有限公司。

本文件主要起草人：高艳、顿胜堡、陈冠男、赵晓莺、崔腾鹤、王智信、陈少蓓、姜晓天、邢怀飞、贺文林、刘盼、王付生、陈益军、谢恩鹏、赵明、卢剑平、杨益红、孟庆强、李颖祎、邹旭杰、周凯旋、王勇、谢明璞、郭忠武、姜卫平、彭海、王清华、徐遥令、孙思凯、周聘、单华琦、王琦、陈晓锋、殷惠清、谭胜淋。

引　言

随着广电领域对超高清视频内容高效批量重制的需求增加，业界公司相继推出了超高清视频重制算法技术。这些技术为标清和高清视频的超高清重制提供了多样性的技术解决方案。然而，当前的超高清重制方案都是算法与硬件设备绑定的一体机模式或云端服务模式，只能使用一家公司的整体方案。如果想要更换其他公司的方案，就必须重新购买一体机和服务，这增加了不必要的成本。此外，随着专业用户对相关技术的认知不断加深，希望能够兼收并蓄各家公司的算法，以针对不同类型场景下的视频进行择优处理，最大化各个算法的优势。然而，由于各家公司的算法接口不一致，无法满足客户对不同公司算法优势互补和无缝切换的实际使用需求。

为了解决上述问题，本文件描述了超高清视频处理算法接口规范，包括算法任务接口和算法服务接口。标准的制定旨在通过对算法接口的统一化、算法框架流程的标准化以及算法服务的平台化，实现一台设备无缝切换不同算法方案，允许不同的算法模块融合使用。这样一来，用户就能够灵活选择不同公司的算法方案，不再受限于特定硬件设备或服务提供商。同时，通过标准化算法接口，专业用户可以更加方便地实现算法的兼容和优势互补，从而提升视频处理的效果和质量。这一标准的制定对于促进超高清视频处理技术的发展和应用具有重要意义，有助于满足超高清领域对高效批量重制的需求，同时降低了客户的成本，提高了用户体验。

本文件的发布机构提请注意，声明符合本文件时，可能使用以下涉及的相关专利：

——一种数据、视频处理方法及装置（中国专利申请号202211528132.2）；

本文件的发布机构对于该专利的真实性、有效性和范围无任何立场。

该专利持有人已向本文件的发布机构承诺，其愿意同任何申请人在合理且无歧视的条款和条件下，就专利授权许可进行谈判。该专利持有人的声明已在本文件的发布机构备案。相关信息可以通过以下联系方式获得：

联系人：苏京

通讯地址：北京市经济技术开发区地泽路9号

邮政编码：100176

电子邮件：sujing@boe.com.cn

电话：13811947489

网址：https://www.boe.com.cn/

请注意除上述专利外，本文件的某些内容仍可能涉及专利。本文件的发布机构不承担识别专利的责任

超高清视频处理算法接口技术规范

1 范围

本文件描述了超高清视频处理算法接口的定义、组成与分类、数据类型、接口要求等内容。

本文件适用于超高清视频处理算法的接入与应用，也可用于指导超高清视频处理系统与算法包、算法服务的系统集成与开发。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

GB/T 5271.15-2008 信息技术词汇 第15部分：编程语言

3 术语、定义和缩略语

3.1 术语和定义

下列术语和定义适用于本文件。

3.1.1

算法任务 algorithm task

在一个处理器或者多个处理器上以交错方式并发执行的子程序模块集合。

3.1.2

算法服务 algorithm service

对算法任务的流程化管理。

3.1.3

对象 object

存储并维持所指运算效果的这些运算的集合和数据。

[来源:GB/T5271.15-2008,2]

3.1.4

类 class

一种抽象引用数据类型，适用于所指各对象，并为这些对象的实例定义内部结构和一套运算的模板，包含成员变量和成员函数。

[来源:GB/T5271.15-2008,2,有修改]

3.1.5

成员变量 member variable

类的组成部分，由一个标识符、一组数据属性、一个或多个地址和多个数据值组成，占用固定长度的内存，包含数据类型和名称。

3.1.6

成员函数 member function

类的组成部分，通常带有形参，并带着所产生的数据值返回到其启动处的子程序。

3.2 缩略语

下列缩略语适用于本文件。

AI: 人工智能（Artificial Intelligence）
AVS：音视频编码标准（Audio Video Coding Standard）
HDR: 高动态范围（High Dynamic Range）
GPU: 图形处理单元（Graphics Processing Unit）
MPEG:运动图像专家组（Moving Picture Expert Group）
WMV：视窗多媒体视频（Windows Media Video）

4 接口组成与分类

4.1 概述

图 1 超高清视频处理系统接口组成图

图1为超高清视频处理系统接口组成图，功能如下：

a) 算法任务包含但不限于视频解码任务、视频处理任务、视频编码任务；

——视频解码任务对输入的视频文件进行解码，得到视频的媒体数据；

——视频处理任务包含但不限于视频降噪、超分辨率，帧率提升、HDR 等超高清视频处理算法功能，可分为 AI 算法任务和其他非 AI 传统算法任务；

——视频编码任务是对经过视频处理任务处理完成的媒体数据进行编码，输出视频文件。

算法任务间传递的数据包括媒体数据、元数据等。元数据用于记录超高清视频处理算法的名称、版本、参数、厂家名称，以及编解码参数等信息。

b) 算法服务对算法任务进行并发调度，具体功能包括任务创建、任务调度、资源管理、数据同步、时间同步，监控算法任务的返回状态和算法运行日志并作出响应，算法服务间传递的数据包括配置节点和任务流程信息等。

c) 存储/播放：经过超高清视频处理系统输出的视频文件可存储在本地硬盘，或者在终端显示设备上进行渲染播放。

4.2 接口分类

如图1所示，超高清视频处理算法接口分为算法任务接口和算法服务接口：

a) 算法任务接口：调用算法任务函数的接口，以实现具体的算法功能；

b) 算法服务接口：调用算法任务集合的接口，以实现对算法任务的流程化管理。

5 算法接口要求

5.1 算法任务接口

5.1.1 算法任务接口概述

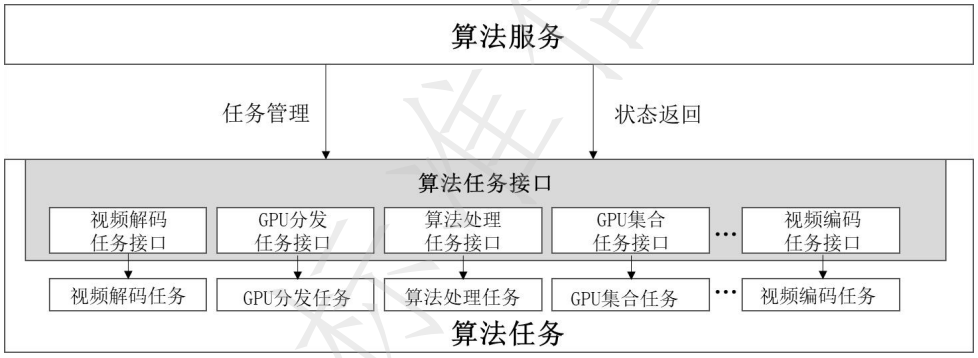

图 2 超高清视频处理算法任务接口示意图

图2为超高清视频处理算法任务接口示意图。算法任务接口包含视频解码任务接口、GPU分发任务接口、视频处理任务接口、GPU集合任务接口、视频编码任务接口，要求如下：

a) 视频解码任接口：通过该接口获取视频解码数据；

b) GPU分发任务接口：通过该接口分发解码帧数据给GPU；

c) 视频处理任务接口：通过该接口对视频帧序列进行算法处理，包含但不限于视频降噪、超分辨率，帧率提升、HDR等视频处理算法功能；

d) GPU集合任务接口：通过该接口将处理后的视频帧数据进行排序；

e) 视频编码任务接口：通过该接口输出编码数据及文件。

5.1.2 算法任务数据类型

5.1.2.1 概述

规定了算法任务接口所传递参数的类型。

5.1.2.2 视频流编码格式类型

视频流编码格式类型应符合表 1 的规定。

表 1 视频流编码格式类型说明表

名称	类型	成员列表	要求
视频流编码格式	枚举	MPEG1VIDEO	必备其中之一
		MPEG2VIDEO	
		MPEG4	
		RAWVIDEO	
		WMV1	
		WMV2	
		H264	
		AVS	
		VP5	
		VP6	
		VP8	
		H265	
		H266	
		VP7	
		AVS2	
		可扩展	

5.1.2.3 视频流像素格式类型

视频流像素格式类型应符合表 2 的规定。

表 2 视频流像素格式类型说明表

名称	类型	成员列表	要求
视频流像素格式	枚举	YUV420P	必备其中之一
		RGB24	
		BGR24	
		YUV422P	
		YUV444P	
		GRAY8	
		PAL8	
		UYVY422	
		NV12	
		NV21	
		ARGB	
		RGBA	
		ABGR	
		BGRA	
		GRAY16BE	
		GRAY16LE	
		YUV440P	
		YUV420P10	
		YUV422P10	
		YUV444P10	
		P010	
		NV24	
		NV42	
		可扩展	

5.1.2.4 视频流色域类型

视频流色域类型应符合表 3 的规定。

表 3 色域类型说明表

名称	类型	成员列表	要求
视频流色域	枚举	RGB	必备其中之一
		BT.709	
		FCC	
		BT.470 BG	
		SMPTE 170 M	
		SMPTE 240 M	
		YCOCG	
		BT.2020 NCL	
		BT.2020 CL	
		SMPTE 2085	
		Chroma-derived NCL	
		Chroma-derived CL	
		ICtCp	
		可扩展	

5.1.2.5 HDR 标准类型

HDR 标准类型应符合表 4 的规定。

表 4 HDR 标准类型说明表

名称	类型	元数据类型	成员列表	要求
HDR标准	枚举	静态元数据	HDR10	必备其中之一
			HLG HDR	
			可扩展	
		动态元数据	HDR10+	
			Dolby Vision	
			HDR Vivid	
			可扩展	

5.1.2.6 视频文件封装类型

视频文件封装类型应符合表 5 的规定。

表 5 视频文件封装类型说明表

名称	类型	成员列表	要求
视频文件封装	枚举	MP4	必备其中之一
		MXF	
		AVI	
		MOV	
		可扩展	

5.1.2.7 视频处理算法标识类型

视频处理算法标识类型应符合表 6 的规定。

表 6 视频处理算法标识类型说明表

名称	类型	成员列表	要求
视频处理算法	枚举	降噪	必备其中之一
		超分	
		插帧	
		HDR	
		可扩展	

5.1.2.8 元数据信息类型

元数据信息类型采用字符串形式，应符合表 7 的规定。

表 7 元数据信息类型说明表

数据类型	成员列表	要求
元数据信息	帧率	必备
	视频编码方式	必备
	色彩空间	必备
	位深	必备
	码率	必备
	像素格式	必备
	HDR标准名称	必备
	视频处理名称	可选
	视频处理版本	可选
	厂家名称	可选
	处理算法名称1	可选
	处理参数1	可选
	处理算法名称2	可选
	处理参数2	可选
	可扩展	-

5.1.2.9 算法任务配置信息类型

算法任务配置信息类型采用字符串形式，应符合表 8 的规定。

超高清视频处理算法任务包含视频解码任务、GPU 分发任务、视频处理任务、GPU 集合任务、视频编码任务等。视频解码任务可对超高清视频文件进行解码，需配置输入文件名称；GPU 分发任务将解码后的视频帧数据进行分发，需配置可调度 GPU 资源；视频处理任务对分发后的帧数据进行处理，需配置的视频处理算法标识；视频处理后的帧数据经过 GPU 集合任务进行有序集合；最后由视频编码任务对集合后的数据进行编码输出，需配置编码器格式、像素格式、封装格式、HDR 标准类型和输出文件名称等。

表 8 算法任务配置信息说明表

数据类型	说明	要求
算法任务配置信息	输入文件名称	必备
	可调度GPU资源名称	必备
	视频处理算法标识[a]	必备
	编码格式名称	必备
	像素格式名称	必备
	封装格式名称	必备
	HDR标准类型名称	必备
	输出文件名称	必备
	可扩展	-
[a]视频处理算法见表6。		

5.1.2.10 视频解码类

视频解码类实现了视频的解码功能，封装了解码视频帧数据接口函数，获取原始输入视频编码信息的接口函数等，应符合表 9 的规定。

表 9 视频解码类主要成员函数说明表

类成员函数	说明	要求
打开视频函数	函数根据视频文件路径或视频流来源解析文件信息，创建解码环境	必备
获取视频帧数据函数	函数从视频文件中解码一帧视频帧数据	必备
获取视频宽函数	返回视频宽	可选
获取视频高函数	返回视频高	可选
获取视频帧率函数	返回视频帧率	可选
获取视频总帧数函数	返回视频总帧数	可选
获取视频封装格式函数	返回视频封装格式信息，类型为结构体	可选
释放资源函数	释放类所占的资源	必备
可扩展	-	-

5.1.2.11 视频编码类

视频编码类实现了视频的编码功能，封装了创建视频文件，视频帧编码等接口函数，应符合表 10 的规定，创建视频文件函数需配置输出视频文件的编码格式和像素格式等，应符合表 11 的规定。

表 10 视频编码类主要成员函数说明表

类成员函数	说明	要求
创建视频文件函数	根据传入的参数创建编码器和输出视频文件，具体参数见表11	必备
设置元数据信息函数	设置编码视频文件所需要的元数据信息，具体参数见 5.1.2.8	可选
视频帧编码函数	编码一帧视频数据，并写入视频文件，输入为视频帧数据内存地址	必备
设置帧率函数	设置视频编码器帧率	必备
设置码率函数	设置视频编码器码率	必备
释放资源函数	释放类所占资源	必备
可扩展	-	-

表 11 创建视频文件函数参数信息

参数名称	输入/输出类型	说明	要求
视频流编码格式类型	输入参数	见5.1.2.2	必备
视频流像素格式类型	输入参数	见5.1.2.3	必备
视频流色域类型	输入参数	见5.1.2.4	可选
HDR标准类型	输入参数	见5.1.2.5	可选
视频文件封装类型	输入参数	见5.1.2.6	必备
视频文件路径	输入参数	输出视频文件路径及文件名	必备
视频文件宽	输入参数	输出视频帧的宽度	可选
视频文件高	输入参数	输出视频帧的高度	可选
可扩展	-	-	-

5.1.3 算法任务接口要求

5.1.3.1 概述

算法任务接口函数返回值若无特殊说明，均为：调用成功，返回 0，调用失败，返回错误码，详细错误码类型见附录 C。

5.1.3.2 视频解码任务接口函数

视频解码任务对输入的视频文件或者视频流等进行解码，接口函数列表应符合表12的规定。

表 12 视频解码任务接口函数列表

接口函数	说明	要求
视频解码任务初始化函数	初始化视频解码任务，将视频解码类传递给视频解码任务；视频解码类详见 5.1.2.10	必备
视频解码任务数据处理函数	视频解码任务的主体，实现了从视频解码类中获取每一帧数据，并把数据传递给其他任务	必备
视频解码任务释放资源函数	释放视频解码任务所占资源，销毁类成员变量	必备
获取视频封装格式信息函数	获取输入视频的封装格式	可选
获取视频帧率函数	获取输入视频的帧率	可选
获取视频总帧数函数	获取输入视频的总帧数	可选
可扩展	-	-

5.1.3.3 视频编码任务接口函数

视频编码任务对视频帧数据进行编码，接口函数列表应符合表13的规定。

表 13 视频编码任务接口函数列表

接口函数	说明	要求
视频编码任务初始化函数	初始化视频编码任务，将视频编码类传递给视频编码任务；视频编码类，详见5.1.2.11	必备
视频编码任务数据处理函数	视频编码任务的主体，实现了编码一帧视频数据	必备
视频编码任务释放资源函数	释放任务所占资源	必备
可扩展	-	-

5.1.3.4 GPU 分发任务接口函数

GPU分发任务按帧号顺序将解码后的视频帧数据分发给不同GPU上的算法任务，接口函数列表应符合表14的规定。

表 14 GPU 分发任务接口函数列表

接口函数	说明	要求
GPU分发任务初始化函数	初始化 GPU 分发任务。函数输入参数为可用 GPU 资源信息。	必备
GPU分发任务数据处理函数	给GPU分发数据	必备
GPU分发任务释放资源函数	释放任务资源	必备
可扩展	-	-

5.1.3.5 GPU 集合任务接口函数

GPU集合任务把多个视频处理任务处理后的视频帧数据按照帧号顺序进行集合，接口函数列表应符合表15的规定。

表 15 GPU 集合任务接口函数列表

接口函数	说明	要求
GPU集合任务初始化函数	初始化GPU集合任务。函数输入参数为可用GPU资源信息。	必备
GPU集合任务数据处理函数	有序集合GPU数据	必备
GPU集合任务释放资源函数	释放任务资源	必备
可扩展	-	-

5.1.3.6 视频处理任务接口函数

视频处理任务对视频帧数据进行算法处理，包括但不限于超分，降噪，插帧，HDR，接口函数列表应符合表16的规定。

表 16 视频处理任务接口函数列表

接口函数	说明	要求
视频处理任务初始化函数	初始化算法任务，函数的输入参数为视频处理算法标识类型，见5.1.2.7	必备
视频处理任务数据处理函数	调用视频处理算法进行数据处理	必备
视频处理任务释放资源函数	释放任务资源	必备
可扩展	-	-

5.1.4 算法任务接口配置流程

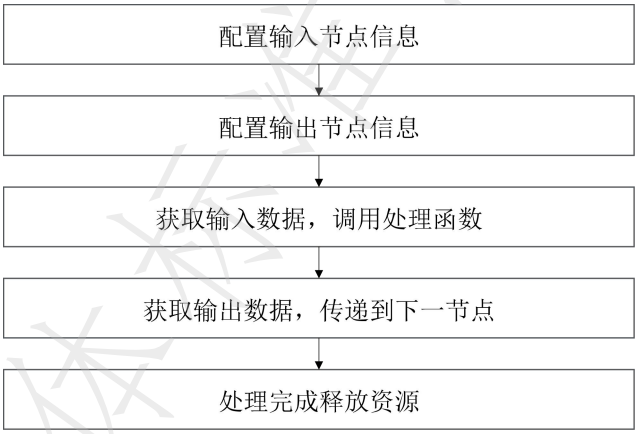

图 3 算法任务接口配置流程

图 3 为算法任务接口配置流程，要求如下：

a) 调用前向节点配置函数配置算法任务输入节点信息；
b) 调用后向节点配置函数配置算法任务输出节点信息；
c) 如上一节点数据可读，获取输入数据，调用数据处理函数对数据进行处理；
d) 若下一节点数据可写，传递数据给下一节点；
e) 处理完成，则安全退出本次算法任务并释放资源。

5.2 算法服务接口

5.2.1 算法服务接口概述

算法服务在超高清处理系统中负责管理算法任务和调度资源，其包括三部分：任务创建类、数据同步类、流程管理类。

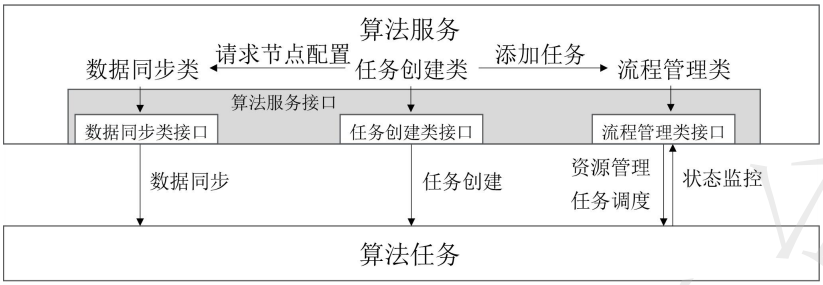

图 4 算法服务接口示意图

图4为算法服务接口示意图，算法服务接口包含数据同步类接口、任务创建类接口、流程管理类接口，这三个类接口实现了对算法任务的流程化管理，要求如下：

a) 调用数据同步类接口可实现算法任务的数据及时间同步传递；

b) 调用任务创建类接口可实现对算法任务规则的预定义；

c) 调用流程管理类接口实现了资源管理、任务调度和状态监控。

5.2.2 算法服务数据类型

5.2.2.1 概述

规定了算法服务接口所传递参数的类型。

5.2.2.2 数据节点类

数据节点类封装了操作数据的接口函数，主要成员应符合表17的规定。

表 17 数据节点类主要成员说明表

成员类型	名称	说明	要求
成员变量	数据链表	存储了数据的具体内容，以查找表的形式定义	必备
	可扩展	-	-
成员函数	初始化函数	根据传入的参数创建数据节点类对象	必备
	获取数据函数	可根据关键字查找对应的数据，函数返回为NULL表示失败，否则为对应数据内存地址	必备
	增加数据函数	可增加数据链表中的数据	必备
	更改数据函数	可根据关键字更改对应的数据，函数返回为NULL表示失败，否则为更改前的数据内存地址	必备
	移除数据函数	可根据关键字移除数据链表中的数据，函数返回表示NULL为失败，否则为移除前的数据内存地址	必备
	可扩展	-	-

5.2.3 算法服务接口要求

5.2.3.1 概述

若无特殊说明，算法服务接口函数的状态返回值均为：调用成功，返回 0，调用失败，返回错误码，详细错误码类型见附录 C。

5.2.3.2 数据同步类

数据同步类实现了算法任务的同步传递机制，包括数据同步和时间同步，数据同步类主要成员应符合表18的规定。

表 18 数据同步类主要成员说明表

成员类型	名称	说明	要求
成员变量	可写信号量	在多线程环境下，用来保证可写内存不被并发调用，类型为信号量，该值大于 0 时，表示数据可写，该值小于 0 时，表示数据不可写。	必备
	可读信号量	在多线程环境下，用来保证可读内存不被并发调用，类型为信号量，该值大于0时，表示数据可读，该值小于0时，表示数据不可读	必备
	数据节点类	见5.2.2.2	必备
	可扩展	-	-
成员函数	初始化函数	创建类的成员变量，并初始化赋值。	必备
	释放资源函数	销毁并释放同步类的成员变量	必备
	可扩展	-	-

图5为数据同步类原理机制说明，规定如下：

数据同步采用信号量来做同步控制，等待信号量时会阻塞当前线程，直到信号量的值大于0继续执行；释放信号量表示会将信号量的值增加1。

等待内存片段0的可读信号量大于0，读取数据；之后释放内存片段0的可写信号量，与此同时上一算法任务的数据可写入内存片段0； 当前算法任务对数据进行处理，等待内存片段1的可写信号量大于0，写入数据；之后释放内存片段1的可读信号量，通知下一算法任务可从内存片段1读取数据。

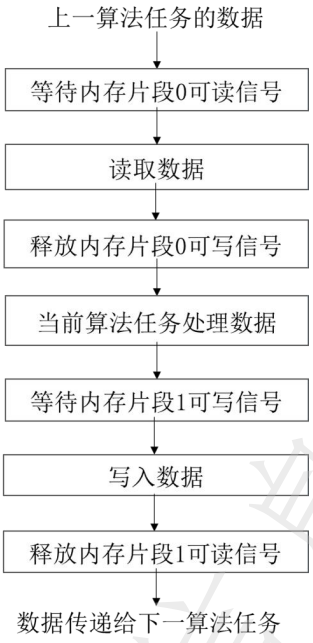

图 5 数据同步类原理机制

5.2.3.3 任务创建类

任务创建类是算法任务创建的预定义规则，算法任务基于此规则创建后，可添加至算法服务中进行流程化管理，任务创建类主要成员应符合表 19 的规定。

表 19 任务创建类主要成员说明表

成员类型	名称	说明	要求
成员变量	前向同步节点	类型为数据同步类，见表18	必备
	后向同步节点	类型为数据同步类，见表18	必备
	前向节点数据	类型为数据节点类，见表17	必备
	后向节点数据	类型为数据节点类，见表17	必备
	可扩展		
成员函数	数据处理函数	实现算法任务的具体操作，为算法任务创建必须重载实现的函数	必备
	资源释放函数	释放算法任务的资源，为算法任务创建必须重载实现的函数	必备
	前向同步节点配置函数	设置算法任务的前向同步节点，为函数参数类型为数据同步类	必备
	后向同步节点配置函数	设置算法任务的后向同步节点，函数参数类型为数据同步类	必备
	数据等待函数	等待上一算法任务的数据，函数参数类型为数据节点类	必备
	数据传递函数	将数据传递给下一算法任务，函数参数类型为数据节点类	必备
	可扩展	–	–

5.2.3.4 流程管理类

流程管理类实现对算法任务的流程化管理，可为算法任务分配并管理资源、调度算法任务，检测算法任务状态，流程管理类主要成员应符合表20的规定。

表 20 流程管理类主要成员说明表

成员类型	名称	说明	要求
成员变量	算法任务存储表	类成员变量，存储视频处理列表内算法任务的链表	必备
	可扩展	-	-
成员函数	初始化函数	初始化视频处理流程管理框架类所占资源	必备
	添加任务函数	算法任务添加至视频处理列表中，函数参数为指向算法任务的指针	必备
	算法任务线程函数	创建算法任务的线程函数	必备
	启动任务函数	创建线程并启动算法任务	必备
	关闭任务函数	关闭视频处理列表内的算法任务，并释放算法任务所占资源	必备
	算法任务存储表	类成员变量，存储视频处理列表内算法任务的链表	必备
	可扩展	-	-

5.2.4 算法服务接口配置流程

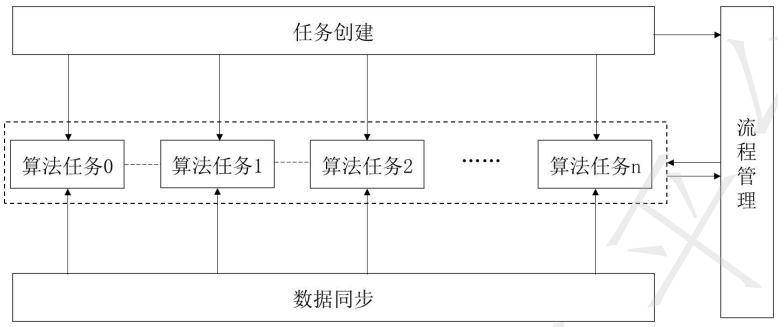

图 5 算法服务接口参考执行流程

图5为算法服务接口参考执行流程，规定如下：

a) 创建算法任务，即图中算法任务 0、算法任务 1、算法任务 2...算法任务 n；
b) 将算法任务添加至流程管理类中进行管理，并为算法任务创建线程，分配运算资源；
c) 创建算法任务间传递对象，并调用配置前向/后向节点函数进行算法任务节点配置；
d) 流程管理类监测算法任务输出节点数据为空时，关闭算法任务并释放资源；
e) 若传入数据为空，则安全退出本次算法任务并释放资源。

5.3 接口补充说明

接口补充说明如下：

a) 接口函数以动态链接库的形式发布，开放给用户使用；
b) 宜提供 Windows 系统、Linux 系统、MacOS 系统，对于不同的操作系统，编译成不同的动态链接库；
c) 内存管理在具体的算法任务中实现；
d) 附录 A 为音频处理任务数据类型和接口规范说明；
e) 附录 B 为函数原型说明和 c++语言调用示例；
f) 附录 C 为函数返回错误码详细说明；
g) 附录 D 为依赖库版本详细说明。

附 录 A

附 录 B（资料性）

附 录 C音频处理任务数据类型和接口规范

C.1 音频处理任务数据类型

A.1.1 音频编码格式类型

音频编码格式类型应符合表 A.1 的规定。

表 A.1 音频编码格式类型说明表

名称	类型	枚举成员列表
音频编码格式	枚举	AAC
		PCM
		可扩展

A.1.2 音频解码类

音频解码类封装了获取音频帧数据的接口函数，应符合表 A.2 的规定。

表 A.2 音频解码类主要成员函数说明表

类成员函数	说明
打开音频函数	创建音频解码环境
获取音频帧数据函数	解码一帧音频数据
释放资源函数	释放类所占的资源

A.1.2 音频编码类

音频编码类封装了编码音频帧数据的接口函数，应符合表 A.3 的规定。

A.3 音频编码类主要成员函数说明表

类成员函数	说明
创建音频函数	根据传入的参数创建音频编码环境，具体参数见表A.1
音频帧编码函数	编码一帧音频数据
释放资源函数	释放类所占资源

C.2 音频处理任务接口

C.2.1 概述

音频处理任务接口函数返回值若无特殊说明，均为：调用成功，返回0，调用失败，返回错误码，详细错误码类型见附录C。

C.2.2 音频解码任务类接口

音频解码任务对输入的音频流进行解码，接口函数列表应符合表A.4的规定。

表 A.4 音频解码任务接口函数列表

接口函数	说明
音频解码任务初始化函数	初始化音频解码任务，将音频解码类传递给音频解码任务；音频解码类详见表 A.2
音频解码任务数据处理函数	音频解码任务的主体，实现了从音频解码类中获取每一帧数据，并把数据传递给其他任务
音频解码任务释放资源函数	释放音频解码任务所占资源，销毁类成员变量

C.2.3 音频编码任务类接口

音频编码任务将音频数据进行编码，接口函数列表应符合表A.5的规定。

表 A.5 音频编码任务接口函数列表

接口函数	说明
音频编码任务初始化函数	初始化音频编码任务，将音频编码类传递给音频编码任务；音频编码类，详见A.3
音频编码任务数据处理函数	是音频编码任务的主体，实现了编码一帧音频数据
音频编码任务资源释放函数	释放任务资源

附 录 D

附 录 E（资料性）

附 录 F数据类型和接口函数原型

B.1 算法服务接口函数

F.1.1 数据同步类

F.1.1.1 函数声明

```
class UnitSynArg{
    public:
        init();//初始化函数
        int release();//资源释放函数
        sem_t readable_sem;//可读信号量
        sem_t writeable_sem;//可写信号量
        PassingPara *passing_para;//数据节点类
};
```

F.1.1.2 数据节点类

F.1.1.2.1 函数声明

```
class PassingPara{
    public:
        std::map<const char *, void *, cmp_str> para_dict;//数据链表
        int init(PassingPara *passing_para);//初始化函数
        void *get_item(const char *);//获取数据函数
        int add_item(const char *, void *);//增加数据函数
        void *change_item(const char *, void *);//更改数据函数
        void *remove_item(const char *);//移除数据函数
};
```

F.1.1.2.2 初始化函数

函数原型：

```
int PassingPara::init(PassingPara *passing_para) ;
```

函数调用参数见表 B.1。

表 B.1 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
passing_para	PassingPara *	输入参数	要复制的数据节点对象

F.1.1.2.3 获取数据函数

函数原型：

```
void *PassingPara::get_item(const char *key);
```

函数调用参数见表 B.2。

表 B.2 获取数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	数据查找的关键字

F.1.1.2.4 增加数据函数

函数原型：

```
int PassingPara::add_item(const char *key, void *val);
```

增加数据函数调用参数见表 B.3。

表 B.3 增加数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	数据查找的关键字
val	void *	输入参数	内存数据的地址指针

F.1.1.2.5 更改数据函数

函数原型：

```
void *PassingPara::change_item(const char *key, void *val);
```

函数调用参数见表 B.4。

表 B.4 更改数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	数据查找的关键字
val	void *	输入参数	内存数据的地址指针

F.1.1.2.6 删除数据函数

函数原型：

```
void *PassingPara::remove_item(const char *key) ;
```

函数调用参数见表 B.5。

表 B.5 删除数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
key	const char *	输入参数	数据查找的关键字

F.1.2 流程管理类

F.1.2.1 函数声明

```
class VideoProcessFramework {
    public:
        int init();//初始化函数
        int add_task(FrameProcessUnit *frame_process_unit);//添加任务函数
        int start_tasks();//启动任务函数
        int close_tasks();//关闭任务函数
    private:
        static void* process_thread(void *arg);//算法任务线程函数
        std::vector<FrameProcessUnit *> process_list;//算法任务存储列表
};
```

F.1.2.2 添加任务函数

函数原型：

```
nt VideoProcessFramework::add_task(FrameProcessUnit *frame_process_unit);
```

函数调用参数见表 B.6

表 B.6 添加任务函数调用参数

参数	数据类型	输入/输出类型	参数说明
frame_process_unit	FrameProcessUnit *	输入参数	指向算法任务单元的指针

F.1.3 任务创建类

F.1.3.1 函数声明

```
class FrameProcessUnit{
    public:
        int pre_node(std::vector<UnitSynArg *> in_nodes);//前向节点配置函数
        int next_node(std::vector<UnitSynArg *> out_nodes);//后向节点配置函数
        int wait(int index, PassingPara **para_temp);//数据等待函数
        int post(int index, PassingPara *para_temp);//数据传递函数
        virtual int process(){};//数据处理函数
        virtual int close(){};//资源释放函数
    private:
        std::vector<UnitSynArg *> syn_in;//前向的同步节点
        std::vector<PassingPara *> para_in;//读取前向的数据传递
        std::vector<UnitSynArg *> syn_out; //后向的同步节点
        std::vector<PassingPara *> para_out;//读取后向的数据传递
};
```

F.1.3.2 前向节点配置函数

函数原型：

```
int FrameProcessUnit::pre_node(std::vector<UnitSynArg *> in_nodes);
```

函数调用参数见表 B.7。

表 B.7　前向节点配置函数调用参数

参数	数据类型	输入/输出类型	参数说明
in_nodes	std::vector<UnitSynArg *>	输入参数	设置算法任务的前向节点

F.1.3.3　后向节点配置函数

函数原型：

```
int FrameProcessUnit::next_node(std::vector<UnitSynArg *> out_nodes);
```

函数调用参数见表B.8。

表 B.8　后向节点配置函数调用参数

参数	数据类型	输入/输出类型	参数说明
in_nodes	std::vector<UnitSynArg *>	输入参数	设置算法任务的后向节点

F.1.3.4　数据等待函数

函数原型：

```
int FrameProcessUnit::wait(int index, PassingPara **para_temp);
```

函数调用参数见表 B.9 。

表 B.9　数据等待函数调用参数

参数	数据类型	输入/输出类型	参数说明
index	int	输入参数	算法任务前向数据节点的索引号
para_temp	PassingPara **	输入/输出参数	算法任务前向数据节点指针的指针

F.1.3.5　数据传递函数

函数原型：

```
int FrameProcessUnit::post(int index, PassingPara *para_temp);
```

函数调用参数见表 B.10。

表 B.10　数据传递函数调用参数

参数	数据类型	输入/输出类型	参数说明
index	int	输入参数	算法任务后向数据节点的索引号
para_temp	PassingPara **	输入/输出参数	算法任务后向数据节点指针的指针

F.2 算法任务接口函数

F.2.1 音视频解码类

F.2.1.1 函数声明

```
class VideoReader{
    public:
        int open_input(const char *filename);
        int read_frame(void *&data_out);
        int getwidth();
        int getheight();
        void get_fps(int &den, int &num);
        void get_nb_frames(int &nbframes);
        AVFormatContext *get_ifmt_ctx();
        int close();
    private:
        //视频的上下文信息
        AVFormatContext *ifmt_ctx = NULL;
        //解码帧
        AVFrame *frame = NULL;
        //媒体流
        AVStream *stream = NULL;
        //视频媒体流的编号
        int video_stream_index = -1;//添加视频流的index
        int audio_stream_index = -1;//添加音频流的index
        //编码上下文
        AVCodecContext *codec_ctx = NULL;
        //帧格式转换上下文
        SwsContext *swsctx = NULL;
        //帧转换输出解码帧
        AVFrame *out_frame = NULL;
        //输出的像素格式
        AVPixelFormat outpixfmt;
        //视频宽
        int width = 0;
        //视频高
        int height = 0;
        //输出图像的宽
        int output_width = 0;
        //输出图像的高
        int output_height = 0;
        //视频像素格式对应的每个像素字节数
        int pixbyte = 0;
};
```

F.2.1.2 打开输入视频函数

函数原型：

```
int VideoReader::open_input(const char *filename);
```

函数调用参数见表B.11。

表 B.11 打开输入视频函数调用参数

参数	数据类型	输入/输出类型	参数说明
filename	const char *	输入	视频文件路径/视频流

F.2.1.3 获取视频帧数据函数

函数原型：

```
int VideoReader::read_frame(void *&data_out);
```

获取视频帧数据函数调用参数见表B.12。

表 B.12 获取视频帧数据函数调用参数

参数	数据类型	输入/输出类型	参数说明
data_out	void *&	输入/输出	视频/音频解码后的数据

F.2.2 音视频编码类

F.2.2.1 函数声明

```
class VideoWriter{
    public:
        typedef struct metadataParam {
            char *key;
            metadata_ID value;
        } metadataParam;//元数据结构体
        //设置元数据信息函数
         int set_metadata_param(metadataParam* metadata_param);
        //创建视频文件函数
        int open_video(const char *filename,int width, int height,
                       Filefmt_ID file_id=0,
                       Codec_ID codec_id=0,
                       Pixfmt_ID pixfmt_id=0,
                       Hdr_ID hdr_id=0,
                       aCodec_ID acodec_id=0,
                       AVFormatContext *ifmt_ctx = NULL);
        int write_videoframe(uint8_t *frame);//视频帧编码函数
```

```
    int write_audiopacke(AVPacket *avpkt);//音频帧编码函数
    int flush();//刷帧函数
    int set_fps(int den, int num);//设置帧率函数
    int set_bitRate(int rate);//设置码率函数
    int close();//释放资源函数
private:
    std::vector<metadataParam *> metadata_param_list;//存储元数据信息的列表
    //输出文件上下文
    AVFormatContext* pFormatCtx = NULL;
    //输入文件上下文
    AVFormatContext *ifmt_ctx = NULL;
    //输出文件封装格式
    AVOutputFormat* fmt = NULL;
    //输出的视频流
    AVStream* video_st = NULL;
    AVStream* video_st_in = NULL;
    AVStream* audio_st_in = NULL;
    AVStream* audio_st_out = NULL;
    int video_stream_index = -1;
    int audio_stream_index = -1;
    //编码上下文
    AVCodecContext* pCodecCtx = NULL;
    AVCodecContext* pAudioDeCodecCtx = NULL;
    AVCodecContext* pAudioEnCodecCtx = NULL;
    //编码器
    AVCodec* pCodec = NULL;
    AVCodec* pAudioDeCodec = NULL;
    AVCodec* pAudioEnCodec = NULL;
    //编码帧
    AVPacket* packet = NULL;
    //原始数据帧
    AVFrame* pFrame = NULL;
    //帧格式转换上下文
    SwsContext *swsctx= NULL;
    //编码器名字
    const char *codec_name = NULL;
    //编码ID
    AVCodecID av_codec_id;
    AVCodecID av_audio_codec_id;
    //像素格式
    AVPixelFormat config_pixfmt;
    //视频的metadata
    AVDictionary *dict = 0;
    //写锁
```

```
        pthread_mutex_t write_lock;
        //帧数统计
        long int frame_cnt = 0;
        //视频的宽
        int width = 0;
        //视频的高
        int height = 0;
        //帧率
        int fps_den = 1;
        int fps_num = 25;
        //码率
        long int bit_rate = 100000000;
        //hdr模式开关
        int hdr_on = 0;
        //创建输出视频文件函数
        int open_output_file(const char *filename, int widht, int height);
        //音频重采样编码相关ffmpeg函数
        SwrContext *audio_resampler_context = NULL;
        AVAudioFifo *audio_fifo = NULL;
        int init_fifo(AVAudioFifo **fifo, AVCodecContext *output_codec_context);
        int convert_samples(const uint8_t **input_data,
                            uint8_t **converted_data, const int frame_size,
                            SwrContext *resample_context);
        int init_resampler(AVCodecContext *input_codec_context,
                           AVCodecContext *output_codec_context,
                           SwrContext **resample_context);
        init_converted_samples(uint8_t ***converted_input_samples,
                               AVCodecContext *output_codec_context,
                               int frame_size);
        add_samples_to_fifo(AVAudioFifo *fifo,
                            uint8_t **converted_input_samples,
                            const int frame_size);
};
```

F.2.2.2 创建视频文件函数

函数原型：

```
int VideoWriter::open_video(const char *filename,int width, int height,
                            Filefmt_ID file_id=0,
                            Codec_ID codec_id=0,
                            Pixfmt_ID pixfmt_id=0,
                            Hdr_ID hdr_id=0,
                            aCodec_ID acodec_id=0,
                            AVFormatContext *ifmt_ctx = NULL);
```

函数调用参数见表A.13。

表 B.13　创建视频文件函数调用参数

参数	数据类型	输入/输出类型	参数说明
filename	const char *	输入	输出视频文件的路径和文件名
width	int	输入	输出视频文件的宽度
height	int	输入	输出视频文件的高度
file_id	Filefmt_ID	输入	输出视频文件的封装格式，默认为 mp4
codec_id	Codec_ID	输入	视频编码器的类型，默认为视频 H264
pixfmt	Pixfmt_ID	输入	视频编码帧的像素格式，默认为 YUV420P
hdr_id	Hdr_ID	输入	视频文件 HDR 标准类型，默认为无 HDR
acodec_id	aCodec_ID	输入	音频编码器的类型，默认为 AAC
ifmt_ctx	AVFormatContext *	输入	源视频文件的封装环境，可以选择性的为输出视频文件对应的封装环境赋值

F.2.2.3　视频帧编码函数

函数原型：

```
int VideoWriter::write_videoframe(uint8_t *frame);
```

函数调用参数见表B.14。

表 B.14　视频帧编码函数调用参数

参数	数据类型	输入/输出类型	参数说明
frame	uint8_t *	输入	视频帧数据指针

F.2.2.4　音频帧编码函数

函数原型：

int VideoWriter::write_audiopacket(AVPacket *avpkt);

函数调用参数见表B.15。

表 B.15 音频帧编码函数调用参数

参数	数据类型	输入/输出类型	参数说明
avpkt	AVPacket *	输入	音频数据包

F.2.2.5 设置帧率函数

函数原型：

int VideoWriter::set_fps(int den, int num) ;

函数调用参数见表B.16。

表 B.16 设置帧率函数调用参数

参数	数据类型	输入/输出类型	参数说明
den	int	输入	时间基的分子
num	int	输入	时间基的分母

F.2.2.6 设置码率函数

函数原型：

int VideoWriter::set_bitRate(long int bit_rate);

函数调用参数见表B.17。

表 B.17 设置码率函数调用参数

参数	数据类型	输入/输出类型	参数说明
bit_rate	long int	输入	输出视频文件的码率

F.2.2.7 设置元数据信息函数

函数原型：

int set_metadata_param(metadataParam* metadata_param);

函数调用参数见表B.18。

表 B.18 设置元数据信息函数调用参数

参数	数据类型	输入/输出类型	参数说明
metadata_param	metadataParam*	输入	指向元数据信息结构体的指针

F.2.3 音视频解码任务

F.2.3.1 函数声明

class VideoReaderUnit : public FrameProcessUnit{

```
    public:
        int init(const char *input_path);
        AVFormatContext *get_ifmt_ctx();
        int process() override;
        int close() override;
        int get_fps(int &den, int &num);
        int get_nb_frames(int &nbframes);
        VideoReader video_reader;
};
```

F.2.3.2 初始化函数

函数原型：

```
int VideoReaderUnit::init(const char *input_path）;
```

函数调用参数见表B.19。

表 B.19 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
input_path	const char *	输入	视频文件的地址和文件名

F.2.3.3 获取视频帧率函数

函数原型：

```
int VideoReaderUnit::get_fps(int &den, int &num);
```

函数调用参数见表B.20。

表 B.20 获取视频帧率函数调用参数

参数	数据类型	输入/输出类型	参数说明
den	int	输入/输出	时间基的分子
num	int	输入/输出	时间基的分母

F.2.3.4 获取视频总帧数函数

函数原型：

```
int VideoReaderUnit::get_nb_frames(int &nbframes);
```

函数调用参数见表B.21。

表 B.21 获取视频总帧数函数调用参数

参数	数据类型	输入/输出类型	参数说明
nbframes	int &	输入/输出	视频文件的总帧数

F.2.4 视频编码任务

F.2.4.1 函数声明

```
class VideoWriterImgUnit : public FrameProcessUnit{
```

```
    public:
        int init(VideoWriter *video_writer);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        VideoWriter *video_writer;//音视频编码类
};
```

F.2.4.2 初始化函数

函数原型：

```
int VideoWriterImgUnit::init(VideoWriter *video_writer) ;
```

函数调用参数见表B.22。

表 B.22 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
video_writer	VideoWriter *	输入	音视频编码类指针

F.2.5 音频编码任务

F.2.5.1 函数声明

```
class VideoWriterPktUnit : public FrameProcessUnit{
    public:
        int init(VideoWriter *video_writer);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        VideoWriter *video_writer;//音视频编码类
};
```

F.2.5.2 初始化函数

函数原型：

```
int VideoWriterPktUnit::init(VideoWriter *video_writer) ;
```

函数调用参数见表B.23。

表 B.23 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
video_writer	VideoWriter *	输入	音视频编码类指针

F.2.6 GPU分发任务

F.2.6.1 函数声明

```
class OneToManyUnit : public FrameProcessUnit{
    public:
        int init(vector<int> vecdeviceid);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        int numGPU = 0;//总GPU数
        int cnt = 0;//计数
};
```

F.2.6.2 初始化函数

函数原型：

```
int OneToManyUnit::init(vector<int> vecDeviceID)；
```

函数调用参数见表B.24。

表 B.24 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
vecDeviceID	vector<int>	输入	容器中存放 GPU 资源的 ID

F.2.7 GPU集合任务

F.2.7.1 函数声明

```
class ManyToOneUnit : public FrameProcessUnit{
    public:
        int init(vector<int> vecdeviceid);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        int numGPU = 0;//总GPU数
        int cnt = 0;//计数
};
```

F.2.7.2 初始化函数

函数原型：

```
init(vector<int> vecdeviceid);
```

函数调用参数见表B.25。

表 B.25 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
vecDeviceID	vector<int>	输入	容器中存放 GPU 资源的 ID

F.2.8 AI模型处理任务

F.2.8.1 函数声明

```
class modelProcessUnit : public FrameProcessUnit{
    public:
        int init(eumMODEL_ID modelID, int gpu_id);//初始化函数
        int process() override;//数据处理函数
        int close() override;//资源释放函数
    private:
        int getModelConfig(eumMODEL_ID modelID,modelConfigInfo& thisConfig);//获取模
型配置信息函数
       myconfigInfo model_config[MODEL_NUM];//模型配置信息表
       trtfilter *model = NULL;//模型tensorrt推理引擎指针
};
```

F.2.8.2 初始化函数

函数原型：

int modelProcessUnit::init(eumMODEL_ID modelID ,int gpu_id）；

函数调用参数见表B.26。

表 B.26 初始化函数调用参数

参数	数据类型	输入/输出类型	参数说明
modelID	eumMODEL_ID	输入	模型枚举号
gpu_num	int	输入	GPU 的 id 号

F.3 调用示例

调用示例如下：

```
int main(int argc, char *argv[]){
int gpu_num =0;//GPU 数量
cudaSts = cudaGetDeviceCount(&gpu_num)
cudaGetDeviceCount(&gpu_num);
char* pInputVideo = "./test_video/2022-08-09-172550.mov";//输入视频路径和文件名
char* pOutVideo = "./out_video/";//输出视频路径
long int nBitRate = 88888888;//输出视频码率
//输出视频编码信息格式
Codec_ID codec_id = H264;
Pixfmt_ID pixfmt_id = YUV420;
   vector<eumMODEL_ID> vecTrtModelID;//AI 算法模型容器
   vecTrtModelID.push_back(SR_1080);//添加超分模型
   vecTrtModelID.push_back(HDR_2160);//添加 HDR 模型
   int nw = 1920;//输入视频解码目标宽度
   int nh = 1080;//输入视频解码目标高度
```

```
        int nc = 3;//输入视频通道数
        int nscale = 2;//超分模型拉伸倍率
        int ishdr = 1;//是否是 hdr 模式
        gpu_num = vecdeviceid.size();//gpu 数量
    //初始化
    int nw_sr = nw*nscale;//输出视频宽
    int nh_sr = nh*nscale;//输出视频高
    VideoProcessFramework video_process_framework;//创建视频流水线类
   VideoReader*video_reader = new VideoReader(pInputVideo, nw, nh, true);//创建音视频解码类
    VideoReaderUnit video_reader_unit;//创建视频解码任务
    sdkSts = video_reader_unit.init(video_reader);
    VideoWriter *video_writer = new VideoWriter();//创建音视频编码类
    int den = 0, num = 0;
    int nbframes;
    video_reader_unit.get_fps(den, num);
    video_reader_unit.get_nb_frames(nbframes);
    video_writer->set_fps(den, num);
    video_writer->set_bitRate(nBitRate);
    sdkSts = video_writer->open_video(pOutVideo,nw_sr, nh_sr,
   codec_id, pixfmt_id, video_reader_unit.get_ifmt_ctx());
    VideoWriterImgUnit video_writer_img_unit;//创建视频编码任务
    video_writer_img_unit.init(video_writer,iscopyfrc,frc,ishdr);
    std::vector<UnitSynArg *> node_videoWriteUnit_in;
    VideoWriterPktUnit video_writer_pkt_unit;//创建音频编码任务
    video_writer_pkt_unit.init(video_writer);
    //视频文件解码后输出视频和音频数据，所以有两个同步节点
    //将两个同步节点打包
   UnitSynArg *synArg_videoReadUint_img_out = new UnitSynArg();//视频数据节点
   UnitSynArg *synArg_videoReadUint_pkt_out = new UnitSynArg();//音频数据节点
    std::vector<UnitSynArg *> videoReadUint_node_out = {synArg_videoReadUint_img_out,
synArg_videoReadUint_pkt_out};
    video_reader_unit.next_node(videoReadUint_node_out);//配置视频解码任务后向节点
    FrameProcessUnit *video_reader_process_unit = &video_reader_unit;//
    video_process_framework.add_task(video_reader_process_unit);//将视频解码任务添加到视频处理流水线中
    std::vector<UnitSynArg *> node_videoPktwWriter_in = {synArg_videoReadUint_pkt_out};
    //配置写音频的入口
    video_writer_pkt_unit.pre_node(node_videoPktwWriter_in);
    //将音频编码任务加入到视频处理流水线
    FrameProcessUnit *video_writer_pkt_process_unit = &video_writer_pkt_unit;
    video_process_framework.add_task(video_writer_pkt_process_unit);
    int model_nums = vecTrtModelID.size();
    //分发器
```

```
    OneToManyUnit one_to_many;
    one_to_many.init(vecdeviceid);
    std::vector<UnitSynArg *> node_oneToManyUint_in = {synArg_videoReadUint_img_out}; //
分发器的入口为 video_reader 类的视频数据出口
    one_to_many.pre_node(node_oneToManyUint_in);
    std::vector<UnitSynArg *> node_oneToManyUint_out;//
    //many_to_one 集成器
    ManyToOneUnit many_to_one;
    many_to_one.init(vecdeviceid);
    std::vector<UnitSynArg *> node_ManyToOneUint_in;//
    std::vector<UnitSynArg *> node_ManyToOneUint_out;//
    //根据 GPU 数量对 AI 算法模型任务进行初始化
    TensorrtProcessUnit tensorrt_model[gpu_num];
   for(int i = 0; i < gpu_num; i++)
   {
        tensorrt_model[i].init(vecTrtModelID, vecdeviceid,i,ishdr);
    }
    //循环每个模型处理模型
    for(int i = 0; i < gpu_num; i++)
    {
        //每个模型一个入口一个出口节点
        UnitSynArg *synArg_trtModelUnit_in = new UnitSynArg();//
        UnitSynArg *synArg_trtModelUnit_out = new UnitSynArg();
        node_oneToManyUint_out.push_back(synArg_trtModelUnit_in);//分发器的出口为多个
gpu,trt 模型单元的入口
        node_ManyToOneUint_in.push_back(synArg_trtModelUnit_out);//集合器的入口为多个
gpu,trt 模型单元的出口
        //配置 AI 算法模型处理任务的入口节点
        std::vector<UnitSynArg *> node_trtModelUint_in = {synArg_trtModelUnit_in};
        tensorrt_model[i].pre_node(node_trtModelUint_in);
        //配置 AI 算法模型处理任务的出口节点
        std::vector<UnitSynArg *> node_trtModelUint_out = {synArg_trtModelUnit_out};
        tensorrt_model[i].next_node(node_trtModelUint_out);
        //将 AI 算法模型处理任务加入视频处理流水线中
        FrameProcessUnit *tensorrt_model_process_unit = &(tensorrt_model[i]);
     video_process_framework.add_task(tensorrt_model_process_unit);
    }
    //将分发任务加入视频处理流水线中
    one_to_many.next_node(node_oneToManyUint_out);
    FrameProcessUnit *one2many_unit = &one_to_many;
    video_process_framework.add_task(one2many_unit);
    //AI 模型任务的出口即为 many_to_one 集成器的入口
    many_to_one.pre_node(node_ManyToOneUint_in);
    //配置模型出口节点
```

```
    UnitSynArg * synArg_mangToOneUint_Out = new UnitSynArg;
    syn_list.push_back(synArg_mangToOneUint_Out);
    node_ManyToOneUint_out.push_back(synArg_mangToOneUint_Out);
    many_to_one.next_node(node_ManyToOneUint_out);
    //模型集合器加入视频处理流水线中
    FrameProcessUnit *many2one_unit = &many_to_one;
    video_process_framework.add_task(many2one_unit);
    node_videoWriteUnit_in.push_back(synArg_mangToOneUint_Out);
    //配置视频编码任务的入口节点
    video_writer_img_unit.pre_node(node_videoWriteUnit_in);
    FrameProcessUnit *video_writer_img_process_unit = &video_writer_img_unit;
    //视频编码任务加入视频处理流水线中
    video_process_framework.add_task(video_writer_img_process_unit);
    //启动任务，等到所有任务结束返回。
    video_process_framework.start_tasks();
    video_process_framework.close_tasks();

}
```

附 录 G

附 录 H（资料性）

附 录 I函数返回错误码详细说明

C.1 返回错误码枚举

算法服务和算法任务接口函数返回错误码以枚举形式列出，详细说明如下。

```
enum mmsdkErrorSts
{
//AVfile
    mmsdkErrorAVOpenInputFileFailed                 = -1000,
    mmsdkErrorAVOpenOutputFileFailed,
    mmsdkErrorAVInputFileReadFinshed,
    mmsdkErrorAVerrorEOF,
    mmsdkErrorAVFileHeaderWriteFailed,
    mmsdkErrorAVFileTrailerWriteFailed,

    //reader
    mmsdkErrorAVVideofillArrayFailed,
    //fmttxt
    mmsdkErrorAVfmtWrong,
    //decoderTxt.encoderTxt
    mmsdkErrorAVAudioCodecTxtParameterCopyFailed,
    mmsdkErrorAVAudioCodecTxtAllocateFailed,
    mmsdkErrorAVVideoCodecTxtParameterCopyFailed,
    mmsdkErrorAVVideoCodecTxtAllocateFailed,
    //decoder
    mmsdkErrorAVAudioDecoderOpenFailed,
    mmsdkErrorAVAudioDecoderFindFailed,
    mmsdkErrorAVVideoDecoderOpenFailed,
    mmsdkErrorAVVideoDecoderFindFailed,
    //encoder
    mmsdkErrorAVAudioEncoderFindFailed,
    mmsdkErrorAVAudioEncoderOpenFailed,
    mmsdkErrorAVVideoEncoderFindFailed,
    mmsdkErrorAVVideoEncoderOpenFailed,
    //stream
    mmsdkErrorAVAudioStreamCreateFailed,
    mmsdkErrorAVAudioStreamInfoFailed,
```

```
    mmsdkErrorAVAudioStreamAllocateFailed,
    mmsdkErrorAVVideoStreamCreateFailed,
    mmsdkErrorAVVideoStreamInfoFailed,
    mmsdkErrorAVVideoStreamAllocateFailed,
    //swr
    mmsdkErrorAVAudioSwrCtxCallocFailed,
    mmsdkErrorAVAudioSwrCtxAllocFailed,
    mmsdkErrorAVAudioSwrCtxInitFailed,
    mmsdkErrorAVAudioSamplesSwrConvertFailed,
    mmsdkErrorAVAudioSwrConvertedSamplesPointerCallocFailed,
    mmsdkErrorAVAudioSwrConvertedSamplesAllocateFailed,
    mmsdkErrorAVVideoSwscaleFailed,
    //packet
    mmsdkErrorAVAudioPacketReadFailed,
    mmsdkErrorAVAudioPacketSendFailed,
    mmsdkErrorAVVideoPacketReadFailed,
    mmsdkErrorAVVideoPacketReceiveFailed,
    mmsdkErrorAVAudioPacketReceiveFailed,
    mmsdkErrorAVVideoPacketSendFailed,
    mmsdkErrorAVAudioAVPacketRescaleTsFailed,
    mmsdkErrorAVVideoAVPacketRescaleTsFailed,
    mmsdkErrorAVVideoPacketInterLeavedWriteFailed,
    mmsdkErrorAVAudioPacketInterLeavedWriteFailed,
    //frame
    mmsdkErrorAVAudioReceiveFrameFailed,
    mmsdkErrorAVVideoReceiveFrameFailed,
    mmsdkErrorAVAudioFrameSendFailed,
    mmsdkErrorAVVideoFrameSendFailed,
    mmsdkErrorVideoFrameSwsScaleFailed,
    mmsdkErrorAVAudioFrameAllocateFailed,
    mmsdkErrorAVVideoFrameAllocateFailed,
    mmsdkErrorAVAudioFrameGetBufferFailed,
    mmsdkErrorAVVideoPtsWorng,
    //audiofifo
    mmsdkErrorAVFifoAllocateFailed,
    mmsdkErrorAVFifoReAllocateFailed,
    mmsdkErrorAVFifoReadFailed,
    mmsdkErrorAVFifoWriteFailed,
    //flush
    mmsdkErrorAVFrameFlushFailed,
    mmsdkErrorAVPacketFlushFailed,
//CUDA
    mmsdkErrorCUDADeviceFailed              = -800,
    mmsdkErrorCUDAMallocFailed,
```

```
    mmsdkErrorCUDAMemcpyDeviceToHostFailed,
    mmsdkErrorCUDAMemcpyHostToDeviceFailed,
    mmsdkErrorCUDAMemcpyDeviceToDeviceFailed,
//Filter Unit
    mmsdkErrorUnitWaitWorng                 = -700,
    mmsdkErrorUnitPostWorng,
    mmsdkErrorParaAdditemWorng,
    mmsdkErrorUnitVideoReadExit,
    mmsdkErrorUnitImgWriterExit,
    mmsdkErrorUnitAudioWriterExit,
    mmsdkErrorUnitTrtModelExit,
    mmsdkErrorUnitOneToMannyExit,
    mmsdkErrorUnitMannyToOneExit,
    mmsdkErrorUnitHostToDeviceExit,
    mmsdkErrorUnitDeviceToHostExit,
    mmsdkSuccess = 0
};
```

I.1 错误码说明函数

函数原型：

```
    char* mmsdkErrorToStr(mmsdkErrorSts threadErr);
```

函数调用参数见表C.1。

表 C.1 错误码说明函数调用参数

参数	数据类型	输入/输出类型	参数说明
threadErr	mmsdkErrorSts	输入	函数返回错误码枚举号

附 录 J

附 录 K（资料性）

附 录 L依赖库版本详细说明

L.1 依赖库版本说明

第三方依赖库版本详细说明见表D.1

表 D.1 第三方依赖库版本说明表

依赖库名称	版本号
ffmpeg	4.3.2
cuda	11.2
cudnn	8.0